TP7 : Analyse lexicale et syntaxique

Objectifs du TP :

- Implémenter un analyseur lexical et syntaxique pour des expressions arithmétiques.
- Faire le lien entre grammaire algébrique et structure d'un mot.
- Réviser la notion de récursivité mutuelle.

L'un des nombreux objectifs de l'étude des langages formels est de construire des outils permettant de reconnaître une suite de caractères comme étant un code syntaxiquement correct dans un langage de programmation donné, étape préalable au fait de donner un sens au dit code. Dans ce TP, on propose de faire l'analyse syntaxique non de code mais d'expressions arithmétiques ce qui simplifiera la tâche. Les étapes et concepts de ce TP sont néanmoins génériques.

Partie 0 *Plan de bataille*

Le but final de ce TP est de construire une fonction evalue : string -> int prenant en entrée une chaîne de caractères correspondant à une expression arithmétique et renvoyant la valeur de cette expression. Ces dernières ne feront intervenir que des entiers et ne pourront utiliser que les opérateurs binaires $+, -, \times, /$ et l'opérateur unaire $-$ dans un premier temps. Ainsi, evalue "((-75+9) * (7- 9))" devra renvoyer 132.

Pour ce faire, on passe par plusieurs étapes :

1. Analyse lexicale. Cette étape consiste à découper la chaîne initiale en une liste de *lexèmes* (en anglais, *token*), chaque lexème étant constitué d'un ensemble de caractères formant une unité. C'est dans cette étape que les caractères 7 et 5 accolés seront reconnus comme formant l'unité lexicale 75.

   Dans notre cas, les lexèmes seront donnés par le type suivant :

   ```
   type lexeme = CONST of int | EOF | LPAR | RPAR | PLUS | TIMES | DIV | MINUS
   ```

   Le lexème LPAR désignera toute parenthèse ouvrante, le lexème RPAR toute parenthèse fermante et le lexème EOF la fin de la chaîne analysée. A la fin de cette étape, on disposera ainsi d'une fonction lexeur : string -> lexeme list telle que lexeur "((-75+9) * (7- 9))" renvoit la liste de lexèmes :

   ```
   [LPAR; LPAR; MINUS; CONST 75; PLUS; CONST 9; RPAR; TIMES; LPAR; CONST 7; MINUS;
   CONST 9; RPAR; RPAR; EOF]
   ```

2. Analyse syntaxique. Les lexèmes précédemment identifiés forment les symboles terminaux pour la grammaire G engendrant le langage des expressions arithmétiques correctement formées. A l'aide de cette grammaire, l'analyse syntaxique permet d'associer à une liste de lexèmes (donc à un mot de $L(G)$) son arbre syntaxique — unique si G est non ambiguë.

   Dans notre cas, un arbre syntaxique pour une expression arithmétique aura le type ea :

   ```
   type op_binaire = Plus | Moins | Fois | Div
   type ea = Const of int | Bin of op_binaire*ea*ea | Oppose of ea
   ```

   A l'issue de cette étape on disposera d'une fonction parseur : lexeme list -> ea.

3. Evaluation. Nous travaillerons ici avec une grammaire non ambiguë ce qui assurera l'unicité des arbres syntaxiques construits par l'analyse syntaxique. On ne pourra donc associer qu'une seule sémantique à une chaîne de caractères formant une expression arithmétique. Cette étape vise à déterminer cette sémantique à partir de l'arbre syntaxique calculé par les étapes précédentes.

Partie 1 *Analyse lexicale*

On appelle *chaîne d'expression arithmétique* toute chaîne de caractères ne faisant intervenir que les caractères (,),+,*,-,/, les chiffres et le caractère espace. Par exemple `"45 (+)7-"` est une chaîne d'expression arithmétique mais pas `"45*8*x"`. L'objectif de cette partie est de construire une fonction qui décompose une chaîne d'expression arithmétique en une liste de lexèmes en ignorant les espaces. Si ce n'est pas possible elle renverra l'exception Erreur_lexicale :

```
exception Erreur_lexicale
```

1. Ecrire une fonction isoler_nombre : string -> int -> lexeme*int. Elle prend en entrée une chaîne s et un entier i tel que la lettre à l'indice i dans s existe et est un chiffre. Elle renvoit le couple (CONST n, j) où n est le nombre commençant à la position i dans s et j est l'indice auquel ce nombre finit dans s.
2. Ecrire une fonction premier_lexeme : string -> int -> lexeme*int prenant en entrée une chaîne s et un indice i dans cette chaîne et renvoyant le premier lexème reconnu à partir de l'indice i dans s et l'indice j de s correspondant à l'indice du dernier caractère formant le dit lexème. Si aucun lexème ne peut être reconnu à partir de l'indice i, on lèvera l'exception Erreur_lexicale.
3. En déduire une fonction lexeur : string -> lexeme list transformant une chaîne de caractères en la liste des lexèmes correspondante si la chaîne est une chaîne d'expression arithmétique et levant Erreur_lexicale sinon.

Avant de poursuivre, vérifier que la chaîne de l'exemple est correctement transformée en liste de lexèmes. Vérifier également qu'une chaîne qui n'est pas une chaîne d'expression arithmétique est correctement détectée.

Partie 2 *Analyse syntaxique*

Maintenant que nous avons identifié la liste des lexèmes d'une chaîne d'expression arithmétique, on cherche à savoir si cette dernière est bien formée. Pour ce faire, on se dote de la grammaire G suivante, décrivant les formules arithmétiques syntaxiquement correctes :

$$
\begin{aligned}
S &\to E\,\mathsf{EOF}\\
E &\to \mathsf{CONST}\,n \mid \mathsf{MINUS}\,E \mid \mathsf{LPAR}\,B\,\mathsf{RPAR}\\
B &\to E\,O\,E\\
O &\to \mathsf{PLUS} \mid \mathsf{MINUS} \mid \mathsf{TIMES} \mid \mathsf{DIV}
\end{aligned}
$$

Une remarque sur la règle $E \to \mathsf{CONST}\,n$. Cette dernière signifie que pour tout entier naturel n représentable sur machine, il existe une règle permettant de dériver E en $\mathsf{CONST}\,n$. Comme on ne peut représenter qu'un nombre fini d'entiers sur machine, il y a un nombre fini de règles de ce type et donc la grammaire G a bien un nombre fini de règles.

L'objectif de cette partie est de construire une fonction qui associe à une liste de lexèmes provenant de l'analyse lexicale d'une chaîne d'expression arithmétique son arbre syntaxique selon cette grammaire si il existe et qui lève l'exception Erreur_syntaxique sinon :

```
exception Erreur_syntaxique
```

4. Ecrire trois fonctions de signatures

   parser_E : lexeme list -> ea*lexeme list
   parser_B : lexeme list -> ea*lexeme list
   parser_O : lexeme list -> ea*lexeme list

   telles que parser_X ℓ renvoit l'arbre syntaxique du plus grand préfixe p de ℓ qui est un mot dérivé du non terminal X et le reste de la liste de lexèmes à analyser une fois que ceux utilisés pour former p ont été supprimés de ℓ. Si ce n'est pas possible, Erreur_syntaxique sera levée.

On rappelle que pour définir deux fonctions mutuellement récursive f1 et f2, la syntaxe est la suivante :

```
let rec f1 =
    corps de la fonction f1
and f2 =
    corps de la fonction f2
```

Dans ces conditions, il est possible d'utiliser f1 dans le corps de la fonction f2 et réciproquement.

5. En déduire une fonction parseur : lexeme list -> ea renvoyant l'arbre syntaxique associé à une liste de lexèmes si cette dernière forme un mot engendré par G et levant Erreur_syntaxique sinon. *Remarque : Cette fonction a donc le même comportement que parser_S avec les notations précédentes.*

Partie 3 *Bonus*

6. En exploitant les parties précédentes, écrire une fonction evalue : string -> int permettant d'évaluer une chaîne à condition qu'il s'agisse d'une chaîne d'expression arithmétique correctement formée.

7. En s'inspirant des idées et techniques développées dans les parties précédentes, écrire un analyseur permettant de construire l'arbre syntaxique d'une formule du calcul propositionnel sur les variables $\{x_i \mid i \in \mathbb{N}\}$ donnée sous forme de chaîne de caractères et vérifiant les correspondances suivantes :

Sémantique	$\wedge$	$\vee$	$\Rightarrow$	$\Leftrightarrow$	$\neg$	x_{42}	$\bot$	$\top$
Symboles	/\\	\\/	->	<->	-	x42	F	V

   Attention, \\ est un caractère. La chaîne de caractères "((V /\\x25) -> (x2 <-> -x3))" devra ainsi être associée un arbre syntaxique dont la sémantique est celle de la formule $((\top \wedge x_{25}) \Rightarrow (x_2 \Leftrightarrow \neg x_3))$